Model-Based Testing with Graph Grammars

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

August 14, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process. We describe a model-based testing setup with Graph Grammars. The result is a system for automatic test generation from Graph Grammars. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system is validated using the results of several case-studies.

Contents

Chapter 1

Introduction

In software development projects, often time and budget costs are exceeded. Laird and Brennan [10] investigated in 2006 that 23% of all software projects are canceled before completion. Furthermore, of the completed projects, only 28% are delivered on time with the average project overrunning the budget with 45%. Testing is an important part of software development, because it decreases future maintainance costs [16]. Testing is a complex process and should be done often [19]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of test-cases. However, when the creation of a test suite is done manually, this still leaves room for human error [13]. The process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the ingenuity of single engineers [27].

1.1 Model-based Testing

The existence of an artifact that explicitly encodes the intended behaviour can help mitigate the implications of these problems. Creating an abstract representation or a *model* of the system is an example of such an artifact. What is meant by a model in this report, is the description of the behavior of a system. Moreover, the term model will be often used to describe transition-based notations, such as finite state machines, labelled transition systems and I/O automata. Statecharts such as UML models are not considered in this report.

A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [15]. This implies that the model itself needs validation. Validating the model usually means that the requirements themselves are scrutinised for consistency and completeness [27].

Tools for automatic test generation already exist. The testing tool developed by Axini[1] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool is used in this report and is referred to as Axini Test Manager (ATM). In Utting et al. [27], a taxonomy is done on different model-based testing tools:

[1] http://www.axini.nl/

- TorX [24]: accepts behaviour models such as I/O labelled transition systems. A version of this tool written in Java under continuous development is JTorX [2].
- Spec Explorer[28]: provides a model editing, composition, exploration, and visualization environment within Visual Studio, and can generate offline .NET test suites or execute tests as they are generated (online).
- JUMBL[20]: an academic model-based statistical testing that supports the development of statistical usage-based models using Markov chains, the analysis of models, and the generation of test cases.
- AETG[5]: implements combinatorial testing, where the number of possible combinations of input variables are reduced to a few 'representative' ones.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency or high model complexity. Models that are often used are transition-based, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem with such models is that a larger number of states decreases the model transparency. We think that low model transparency make errors harder to detect and that it obstructs the feedback process of the stakeholders. Using models with high transparency is therefore essential.

1.2 Graph Transformation

A formalism that claims to have more model transparency is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling than traditional state machines. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]:

> Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples.

An informative paper on graph transformations is written by Heckel et al. [8]. A quote from this paper:

> Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular.

The graph transformation tool GROOVE[2] is used to model and explore graph grammars.

1.3 Research goals

The motivation above is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold,

[2] http://sourceforge.net/projects/groove/

this new testing approach will lead to a more efficient testing process and fewer incorrect models. The to be designed system, once implemented and validated, provides a valuable contribution to the testing paradigm. The tools GROOVE and ATM are used to create this system.

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

The solution has to uphold three requirements:

1. A graph grammar must be used as the model; it must derive from the specification and be used for the testing.
2. It must be possible to measure the test progress/completion, by means of *coverage* statistics (explained in detail in section 2.1.4).
3. The solution must be efficient: it should be usable in practice, therefore the technique should be scalable and the imposed constraints reasonable from a practical view point.

1.4 Roadmap

This report features five more chapters: first, the concepts described in this chapter are elaborated in chapter 2. The technique used in GRATiS is described in chapter 3. The setup for the validation of GRATiS is in chapter 5. Chapter 6 features model examples on which GRATiS is applied and validated. Chapter 7 features a case study where GRATiS is applied on an existing software system. Finally, conclusions are drawn in chapter 8.

Chapter 2

Background

The structure of the rest of this chapter is as follows: the general model-based testing process is set out in section 2.1. Some basic concepts from algebra are described in section 2.2. The symbolic models from ATM are then described in section 2.3. Section 2.4 describes the graph grammar formalism. GROOVE and ATM are described in section 2.5.

2.1 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 2.1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses. It gives a 'pass' or 'fail' verdict depending on whether the SUT conforms to the specification or not respectively.

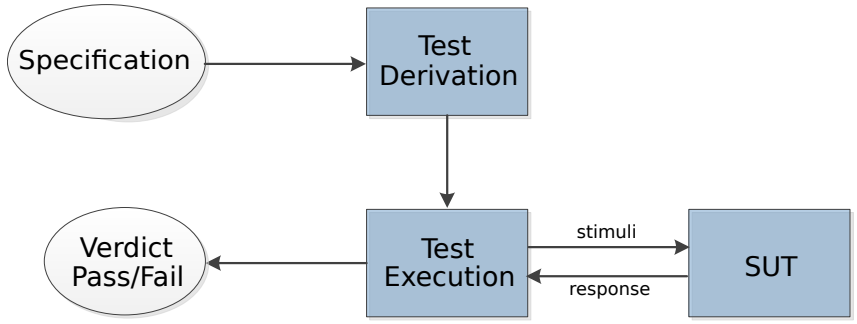


Figure 2.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on-the-fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2.2. An example of an on-the-fly testing tool is TorX [24].

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data.

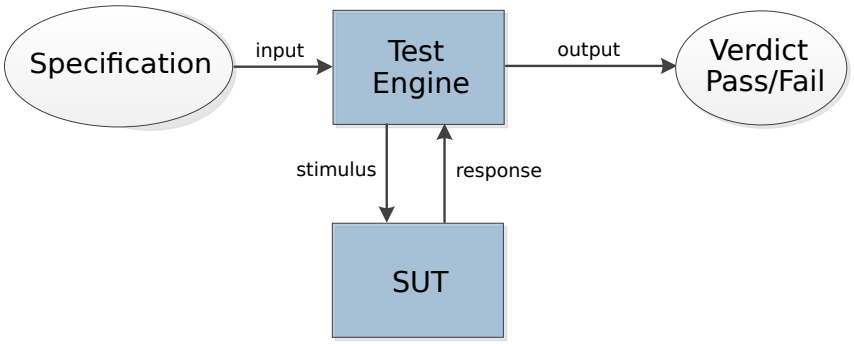


Figure 2.2: A general 'on-the-fly' model-based testing setup

The stucture of the rest of this section is as follows. First, previous work on model-based testing is given. Then, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Finally, the notion of *coverage* is explained.

2.1.1 Previous work

Formal testing theory was introduced by De Nicola et al. [18]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [25] and an algorithm for deriving a sound and exhaustive test suite from a specification in [26]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [14].

2.1.2 Labelled Transition Systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 2.1.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q_0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q_0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

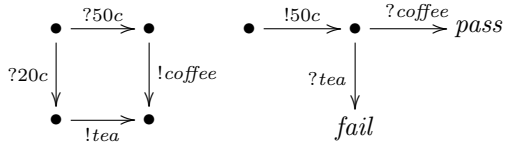
2.1.3 Input-Output Transition Systems

A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [26]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $\iota \in Y$, where $Y =$

$\{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test.

An example of such an IOTS is shown in Figure 2.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a '?', the outputs are preceded by an '!'. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states. A test case for the coffee machine is given in Figure 2.3b. The test case shows that when an input of '50c' is done, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 2.3: The specification of a coffee machine and a test case as an IOTS

2.1.4 Coverage

The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [12, 17, 7].

As an example, let us calculate the coverage metrics of the IOTS test case example in 2.3b. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [29]. These statisics are therefore useful metrics for communicating how much of a system is tested.

2.2 Algebra

Some basic concepts from algebra are described here. For a general introduction into logic we refer to [9].

A *multi-sorted signature* $\langle S, F \rangle$ describes the function symbols and sorts of a formal language. F is a set of function symbols. S is a set of sorts. Each $f \in F$ has an arity $n \in \mathbb{N}$, where a function symbol with arity $n = 0$ is called a constant symbol. F^i denotes the subset of F, with function symbols of arity $n = i$. The sort of a function symbol $f \in F$ with arity n is given by $\sigma(f) = s_1 ... s_n + 1$, with $s_i \in S$ for $1 \leq i \leq n$. S_{n+1} is the return sort. In this report, $S = \{int, real, bool, string\}$ denoting the integer, real, boolean and string sorts respectively. F

features the commonly used function symbols, which include, but not restricted by, '+', '*', '=', '<', '0', '1'.

An *algebra* $\mathcal{A} = \langle \mathbb{U}, \Phi \rangle$ has a non-empty set $\mathbb{U}$ of constants called a *universe*, partitioned into $\mathbb{U}^s$ for each $s \in S$, and a set Φ of functions. A function $\phi_{\mathcal{A}}$ is typed $\mathbb{U}_{\mathcal{A}}^{s_1} \times ... \mathbb{U}_{\mathcal{A}}^{s_n} \to \mathbb{U}_{\mathcal{A}}^{s_{n+1}}$, where $s_1 ... s_{n+1}$ is the sort of the function symbol given by the signature. For example, $<_{\mathcal{A}}$: $\mathbb{U}_{\mathcal{A}}^{int} \times \mathbb{U}_{\mathcal{A}}^{int} \to \mathbb{U}_{\mathcal{A}}^{bool}$ represents the 'less-than' comparison of two integers.

We define $\mathcal{V} = \mathcal{V}^{int} \uplus \mathcal{V}^{real} \uplus \mathcal{V}^{bool} \uplus \mathcal{V}^{string}$ to be the set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from function symbols F and variables $V \subseteq \mathcal{V}$. The definition of a term is:

$$\begin{array}{rcll} t & ::= & f(t_1 ... t_n) & \\ & | & x & , \textit{ where } x \textit{ is a constant.} \end{array}$$

We write $var(t)$ to denote the set of variables appearing in a term $t \in \mathcal{T}(V)$. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms. An example of a term t is $(x + (y - 1))$, with $var(t) = \{x, y\}$. The type of a term is given by:

$$\begin{array}{rlll} \sigma : t \mapsto & s & \textit{if } t = x \in \mathcal{V}^s & \\ & s_{n+1} & \textit{if } t = \phi(t_1 ... t_n) \textit{ and } \sigma(\phi) = s_1 ... s_{n+1}, \textit{ provided } \sigma(t_i) = s_i \end{array}$$

The set of terms with return types $\mathbb{U}^{bool}$, is denoted as $\mathcal{B}(\mathcal{V})$. An example is $(x < y)$, where the result is *true* or *false*.

A *term-mapping* is a function $\mu : \mathcal{V} \to \mathcal{T}(\mathcal{V})$. A *valuation* ν is a function $\nu : \mathcal{V} \to \mathbb{U}$ that assigns values to variables. For example, given an algebra, $\nu : \{(x \mapsto 1), (y \mapsto 2))\}$ assigns the values 1 and 2 to the variables x and y respectively. A valuation of a term given $\mathcal{A}$ is defined by:

$$\begin{array}{rlll} \nu : & x & \mapsto & \nu(x) \\ & (f(t_1 ... t_n)) & \mapsto & f_{\mathcal{A}}(\nu(t_1) ... \nu(t_n)) \end{array}$$

When every variable in a term is defined by a valuation, the term can be valuated to a value. Therefore, when every variable in a term-mapping is defined by a valuation, a new valuation can be obtained. Formally, this is defined as: $\_after\_ : (\mathcal{V} \to \mathbb{U}) \times (\mathcal{V} \to \mathcal{T}(\mathcal{V})) \to (\mathcal{V} \to \mathbb{U})$. Given a valuation ν and a term-mapping μ, $(\nu \textit{ after } \mu) : \nu \mapsto \nu(\mu(\nu))$.

2.3 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. These systems are used in practice in ATM and will therefore be part of GRATiS. In this section, previous work on STSs is given. The definitions of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the transformation of an STS to an LTS is explained and illustrated by an example. This transformation is useful when comparing STSs to systems that are not STSs. Finally, different coverage metrics on STSs are explained.

2.3.1 Previous work

STSs are introduced by Frantzen et al. [11]. This paper includes a detailed definition, on which the definition in section 2.3.2 is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [23]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [11]. However, the definition of IOSTSs we will use in this report is based on the STSs by [11]. A tool that generates tests based on symbolic specifications is the STG tool, described in Clarke et al. [4].

2.3.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term $t \in \mathcal{B}(\mathcal{V})$. The guard disallows using the switch relation when the valuation of the term results in *false*. When the valuation results in *true*, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the valuation of the term.

Definition 2.3.1. A Symbolic Transition System is a tuple $\langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D \rangle$, where:

- W is a finite set of locations and $w_0 \in W$ is the initial location.
- $\mathcal{L} \subseteq \mathcal{V}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \to \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I} \subseteq \mathcal{V}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\Lambda \in \Lambda_\tau$, denoted $arity(\Lambda)$, is a natural number. The parameters of a gate $\Lambda \in \Lambda_\tau$, denoted $param(\Lambda)$, are a tuple of length $arity(\Lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $D \subseteq W \times \Lambda_\tau \times \mathcal{B}(\mathcal{L} \cup \mathcal{I}) \times (\mathcal{L} \to \mathcal{T}(\mathcal{L} \cup \mathcal{I})) \times W$, is the switch relation. We write $w \xrightarrow{\lambda,\gamma,\rho} w'$ instead of $(w, \lambda, \gamma, \rho, w') \in D$, where γ is referred to as the guard and ρ as the update mapping. We require $var(\gamma) \cup var(\rho \subseteq \mathcal{L} \cup param(\lambda))$. We define $out(w) \subseteq D$ to be the outgoing switch relations from location w.

2.3.3 Input-Output Symbolic Transition Systems

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type $\iota \in Y$. As with IOTSs, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively.

2.3.4 Example

In Figure 2.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The values in the tuple of the IOSTS are defined as follows:

$$
\begin{array}{lcl}
W & = & \{t, m\} \\
w_0 & = & t \\
\mathcal{L} & = & \{T, P1, P2, D\} \\
\imath & = & \{T \mapsto 0, P1 \mapsto 0, P2 \mapsto 2, D \mapsto 0\} \\
\mathcal{I} & = & \{d, p, l\} \\
\Lambda & = & \{?throw, !move\} \\
D & = & \{t \xrightarrow{?throw, 1<=d<=6, D \mapsto d} m, \\
 & & \quad m \xrightarrow{!move, T=1 \wedge l=(P1+D)\%4, P1 \mapsto l, T \mapsto 2} t, \\
 & & \quad m \xrightarrow{!move, T=2 \wedge l=(P2+D)\%4, P2 \mapsto l, T \mapsto 1} t\}
\end{array}
$$

The variables $T, P1, P2$ and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $?throws$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throws$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 2.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

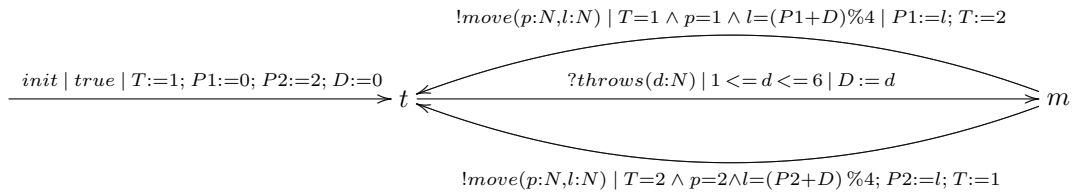


Figure 2.4: The STS of a board game example

2.3.5 STS to LTS mapping

Consider an STS J and an LTS K. There exists a mapping from the location and location variable valuations to the states of K and from the switch relations and variable valuations of J to the transitions of K, such that K is an expansion of J. These relations are defined as follows:

$\mu_Q: \quad (W \times (\mathcal{L} \to \mathbb{U})) \to Q$

$\mu_L: \quad (\Lambda \times (\mathcal{I} \to \mathbb{U})) \to L$

$\mu_T: \quad (w \xrightarrow{\lambda,\gamma,\rho} w', \nu : ((\mathcal{L} \cup \mathcal{I}) \to \mathbb{U})) \mapsto (\mu_Q(w, \nu \restriction \mathcal{L}) \xrightarrow{\mu_L(\lambda,\nu\restriction\mathcal{I})} \mu_Q(w', \nu \; after \; \rho))$

When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ and the number of locations in an STS is considered to be finite, the transformation is always possible to an LTS with finite number of states.

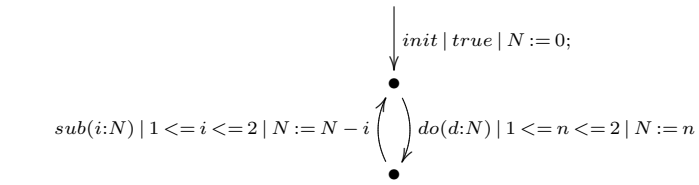
An example of this transformation is shown in Figure 2.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The text on the nodes indicate from which location and valuation the state was created. The node labelled '$w_0, N = 2$' is an example of an unreachable state.

2.3.6 Coverage

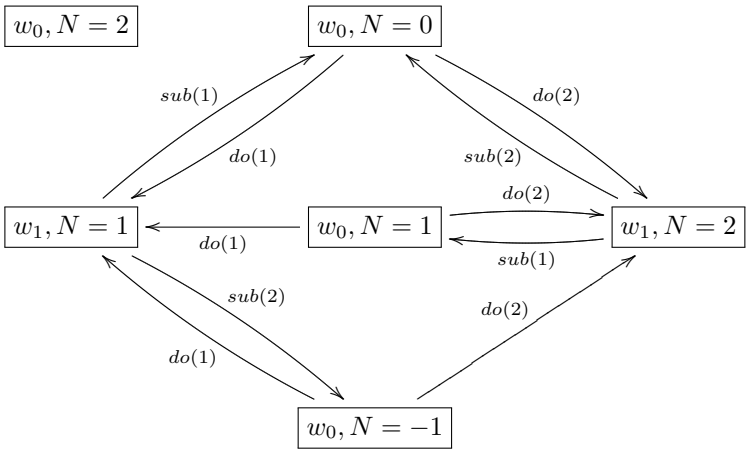
The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring state and transition coverage of an STS is possible using the LTS resulting from the STS transformation.

2.4 Graph Grammars

A *Graph Grammar* (GG) is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.



(a) The STS



(b) The LTS

Figure 2.5: An example of a transformation of an STS to an LTS

The rest of this section is ordered as follows: first, graphs, host graphs, rule graphs and graph transformation rules are explained. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, the definition of IOGGs is given. For a more detailed overview of GGs, we refer to [21, 8, 1].

Definition 2.4.1. A *graph* is composed of nodes and edges. In this report, we assume a universe of nodes $\mathbb{V} = \mathbb{W} \uplus \mathbb{U} \uplus \mathcal{V} \uplus 2^{\mathcal{T}}$, where $\mathbb{W}$ is the universe of standard graph nodes. $\mathbb{E}$ is the universe of edges between two nodes in $\mathbb{V}$.

Definition 2.4.2. A host graph G is a tuple $\langle V_{G^h}, E_{G^h} \rangle$, where:

- $V_{G^h} \subseteq (\mathbb{W} \uplus \mathbb{U})$ is the node set of G
- $E_{G^h} \subseteq (V_{G^h} \backslash \mathbb{U} \times L \times V_{G^h})$ is the edge set of G

Figure 2.6 shows an example of a host graph. Here, $n_1, n_2 \in \mathbb{W}$ are the *identities* of the nodes. The other four nodes are values in $\mathbb{U}$.

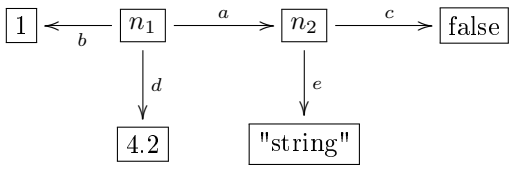


Figure 2.6: An example of a host graph

Definition 2.4.3. A rule graph H is a tuple $\langle V_{G^r}, E_{G^r} \rangle$, where:

- $V_{G^r} \subseteq (\mathbb{V} \backslash \mathbb{U})$ is the node set of H

- $E_{G^r} \subseteq (V_{G^r} \times L \times V_{G^r})$ is the edge set of H

In addition, the following must hold:

- $\forall z \in V_{G^r} \wedge z \in 2^{\mathcal{T}} \,.\, var(z) \subseteq V_{G^r}$ - The variables used in the terms must be present as nodes in the rule graph.
- $(\forall z \in V_{G^r} \wedge z \in \mathcal{V} \,.\, \exists(\_, \_, z) \in E_{G^r}$ - If a variable is used in a rule graph, it needs context. Therefore, there must be an edge with the variable node as target.

Figure 2.7 shows an example of a rule graph. Here, $r_1, r_2 \in \mathbb{W}$ are the node identities, $x_1, x_2 \in \mathcal{V}^{int}$ and $\{x_1 + 1, x_2\} \in 2^{\mathcal{T}}$. The set of terms is mapped as a node to the same value. This mapping is explained in the next definition. The consequence is that this node implicitly expresses the relation $x_1 + 1 = x_2$.

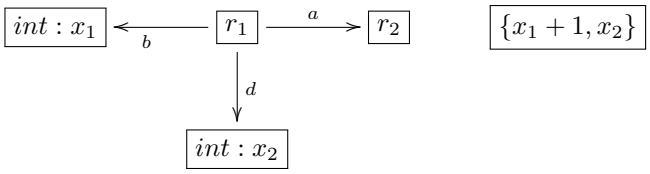


Figure 2.7: An example of a rule graph

Definition 2.4.4. A graph g has a *morphism* to a graph g' if there is a structure-preserving mapping from the nodes and the edges of g to the nodes and the edges of g' respectively. A graph g has a partial morphism to a graph g' if there are elements in g without an image in g'.

Definition 2.4.5. A node or edge z in graph g has an *image* in graph g' and z is a *pre-image* of the image. For each type of node, an explanation is given A variable $v \in \mathcal{V}^s, s \in S$, has an image i in a host graph if $i \in \mathbb{U}^s$. A node $z \in 2^{\mathcal{T}}$ has an image i in a host graph if i is the valuation of all terms in z.

Definition 2.4.6. A transformation rule is a tuple $\langle LHS, NAC, RHS, l \rangle$, where:

- *LHS* is a rule graph representing the left-hand side of the rule
- *NAC* is a set of rule graphs representing the negative application conditions
- *RHS* is a rule graph representing the right-hand side of the rule
- $l \in L$ is the label of the rule

There exist implicit partial morphisms from the *LHS* to each rule graph in *NAC* and from the *LHS* to the *RHS* by means of the node identities. These morphisms are *rule graph morphisms*.

Definition 2.4.7. A *creator* edge is an edge in the *RHS* of a rule, but not in the *LHS* of the rule. An *eraser* edge is an edge in the *LHS* of a rule, but not in the *RHS* or a rule.

Definition 2.4.8. A rule r has a *rule match* on a host graph G if its *LHS* has a morphism in G and $\nexists n \in NAC$ such that n has a morphism in G and $\forall e \in LHS$, if e has an image i in n, and an image j in G, then j should be an image of i. The morphism of the *LHS* to a host graph is a *match morphism*.

Definition 2.4.9. After the rule match is applied to the graph, all elements in *LHS* that do not have an image in *RHS*, are removed from G and all elements in *RHS* that do not have a pre-image in *LHS*, are added to G. This process, called a *rule transition*, is denoted $G \xrightarrow{r,m} G'$, where $m \in M$ is the morphism of the *LHS* to G.

Figure 2.8 shows an example of the initial graph G_0, one rule of a GG and the corresponding rule match. G_0 can be represented by $\langle \{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The *LHS* of

the rule has a match in G_0. Neither *NAC1* and *NAC2* have a match in G_0, because the edge with label C does not exist in G_0. The new graph after applying the rule is G_1.

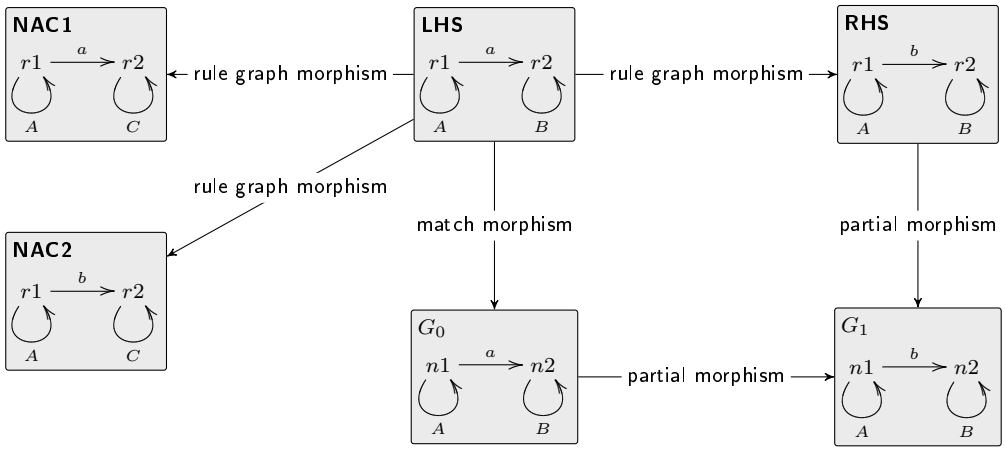


Figure 2.8: An example of a graph transformation

Definition 2.4.10. A graph grammar is a tuple $\langle R, G_0 \rangle$, where:

- R is a set of graph transformation rules
- G_0 is the initial graph

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of graphs connected by rule transitions.

Definition 2.4.11. A graph transition system is a tuple $\langle \mathcal{G}, R, M, U, G_0 \rangle$, where:

- $\mathcal{G}$ is a set of graphs
- $L \in R \times M$ is a set of labels
- $U \in \mathcal{G} \times L \times \mathcal{G}$ is the rule transition relation
- $G_0 \in \mathcal{G}$ is the initial graph

Let $K = \langle R, G_0$. A GTS $O = \langle \mathcal{G}, R, M, U \rangle$ is derived from a K by the following. $\mathcal{G}, M, U$ are the smallest sets, such that:

- $G_0 \in \mathcal{G}$
- if $G \in \mathcal{G}$ and $G \xrightarrow{r,m} G'$ then $G' \in \mathcal{G}, (r, m) \in L, (G \xrightarrow{r,m} G') \in U$

Definition 2.4.12. In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule label $l \in L$ has a type $\iota \in Y$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS). The rule transitions derive their type from their corresponding rule.

2.5 Tooling

2.5.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 2.9. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 2.2.

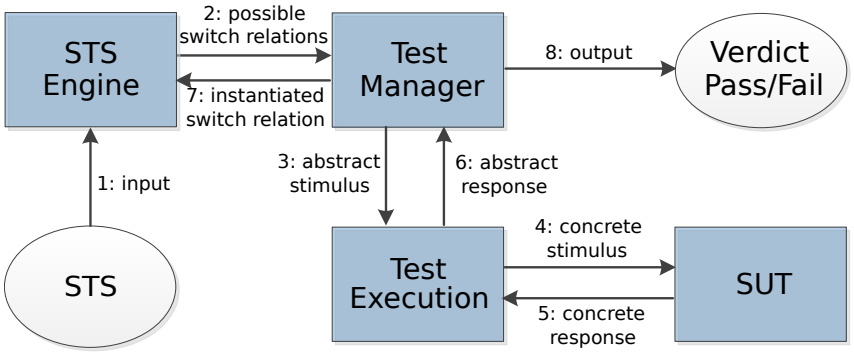


Figure 2.9: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.
2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.
3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label '!ok'. The Test Manager is notified with this abstract response.
5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.